Filière : Licence Génie Informatique
Prof : A. BENMAKHLOUF

Optimisation des Requêtes SQL

Table des matières

I- Traitement d'une requête 2
I-1- La décomposition 2
a- Analyse syntaxique : 2
b- Analyse sémantique : 2
c- Simplification : 2
d- Traduction Algébrique : 2
II- L'Optimisation : Réécriture Algébrique 4
III- L'Optimisation : Le mode d'accès aux données 6
a- Accès Séquentiel : 8
b- Accès Séquentiel Indexé : 9
c- Accès séquentiel indexé avec arbres B 12
d- Accès séquentiel indexé avec le Hachage 17
IV- L'optimisation : Les algorithmes de jointure 23
IV-1 Jointure en boucles imbriquées (Nested Loop) 23
VI-2 Jointure par boucles imbriquées avec indexe 25
IV-3 Jointure par hachage (Hash Loop) 27
V- L'optimisation : Les Vue Matérialisées 30
V-1- Les types de vues matérialisées 30
V-2- Les méthodes de rafraichissement de la VM 31
V-3- Les modes de rafraichissement de la VM 36
VI- Plan d'exécution d'Oracle 37

Introduction

Le temps d'exécution d'une requête SQL dépend d'une part de la structure avec laquelle les opérations algébriques sont écrite et d'autre part du mode d'accès aux données. L'optimisation est donc une opération avec laquelle plusieurs plans d'exécution sont examiné dans l'objectif est de sélectionner celui qui un coût minimum. Ce coût dépend du temps d'exécution et du nombre de ressources utilisées. Il se mesure en nombre d'entrée-Sortie.

Plusieurs SGBD comme Oracle et MySQL possèdent des fonctions permettant d'effectuer ces calculs, via un optimiseur.

I- Traitement d'une requête

L'SQL est un langage déclaratif qui nous permet, à travers un ensemble d'opérations, d'obtenir un ensemble d'information à partir d'une base de données normalisées sans se soucier du comment le SGBD doit obtenir ces informations. Ce dernier, à travers un optimiseur, doit déterminer la façon d'exécuter la requête en choisissant le meilleur plan d'exécution, puis exécuter le plan choisi. Les trois étapes du traitement des requêtes sont la décomposition, l'optimisation et l'évaluation.

I-1- La décomposition

Dans cette étape les opérations suivantes sont effectuées :

a- Analyse syntaxique :

Examiner les caractères saisis et les reconnaître comme des commandes ou des instructions en recherchant des mots clés et des identifiants dans les caractères, en ignorant les commentaires, en organisant les portions entre guillemets en constantes de chaîne et en faisant correspondre la structure globale à la syntaxe du langage.

b- Analyse sémantique :

Vérifier l'exitance dans la BDD des attributs et des tables utilisés dans la requête. Ainsi la détection des incohérences dans les opérations réalisées sur les attributs.

c- Simplification :

Simplifier les expressions logiques utilisées dans les sélections.

example: (A OR non B) AND B = A AND B

d- Traduction Algébrique :

Produire une expression algébrique équivalente à la requête SQL.

Exemple : Soit le schéma relationnel suivant :

Cinéma (IDcinéma, nom, adresse)
Salle (IDsalle, IDcinéma, capacité)
Séance (IDsalle, IDfilm, heuredébut)
Film (IDfilm, réalisateur, année)

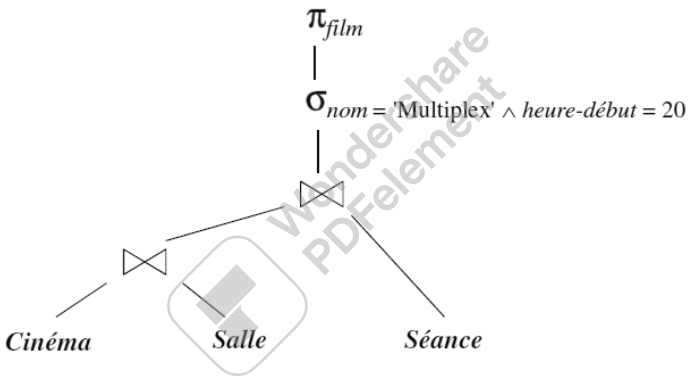
Soit la requête-1 suivante qui donne la liste des films commencent au Multiplex à 20 heures ?

```
SELECT Séance.IDfilm
FROM Cinéma, Salle, Séance
WHERE Cinéma.nom = 'Multiplex' AND
Séance.heure-début = 20 AND
Cinéma.ID-cinéma = Salle.ID-cinéma AND
Salle.ID-salle = Séance.ID-salle
```

L'écriture algébrique de cette requête est donnée par :

$\pi_{film}\ (\sigma_{nom\ =\ 'Multiplex'\ \wedge\ heure\text{-}début\ =\ 20}\ ((Cinéma \bowtie Salle) \bowtie Séance)$

Le plan d'exécution de l'expression algébrique relationnelle est :



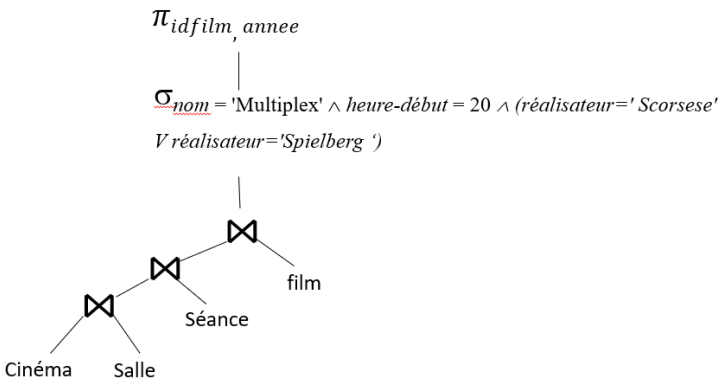
Soit la requête-2 suivante qui donne la liste des films des réalisateurs 'Scorsese' et 'Spielberg' projetés au Multiplex à 20 heures ?

```
SELECT Séance.IDfilm, année
FROM Cinéma, Salle, Séance, film
WHERE Cinéma.nom = 'Multiplex' AND
Séance.heure-début = 20 AND
(réalisateur=' Scorsese' OR réalisateur=' Spielberg ') AND
Cinéma.ID-cinéma = Salle.ID-cinéma AND
Salle.ID-salle = Séance.ID-salle
Séance.IDfilm=film.IDfilm
```

L'écriture algébrique de cette requête est donnée par :

$\pi_{idfilm,\ annee}\ (\sigma_{nom\ =\ 'Multiplex'\ \wedge\ heure\text{-}début\ =\ 20\ \wedge\ (réalisateur='\ Scorsese'\ V\ réalisateur='Spielberg\ ')}$
$(((Cinéma \bowtie Salle) \bowtie Séance) \bowtie film)$

Le plan d'exécution de l'expression algébrique relationnelle est :



II- L'Optimisation : Réécriture Algébrique

Pour une requête il y a plusieurs expressions algébriques équivalentes. Le rôle principal de l'optimiseur est de trouver ces expressions puis évaluer leurs coûts et choisir la meilleure. On peut passer d'une expression à une autre équivalente en utilisant des règles de réécriture. Les plus importantes de ces règles sont :

– Commutativité des jointures
$R \bowtie S \equiv S \bowtie R$

– Associativité des jointures
$(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$

– Regroupement des sélections
$\sigma_{A='a' \wedge B='b'}(R) \equiv \sigma_{A='a'}(\sigma_{B='b'}(R))$

– Commutativité de la sélection et de la projection
$\pi_{A1,\ldots,An}(\sigma_{Ai='a'}(R)) \equiv \sigma_{Ai='a'}(\pi_{A1,\ldots,An}(R))$, $i \in \{1, \ldots, n\}$

– Commutativité de la sélection et de la jointure
$\sigma_{A='a'}(R(\ldots, A, \ldots) \bowtie S) \equiv \sigma_{A='a'}(R) \bowtie S$

– Distributivité de la sélection sur l'union (pareil pour la différence)
$\sigma_{A='a'}(R \cup S) \equiv \sigma_{A='a'}(R) \cup \sigma_{A='a'}(S)$

– Commutativité de la projection et de la jointure
$\pi_{A1,\ldots,An,\ B1,\ \ldots,\ Bm,}(R \bowtie_{Ai = Bj} S) \equiv \pi_{A1,\ldots,An}(R) \bowtie_{Ai = Bj} \pi_{B1,\ \ldots,\ Bm}(S)$

– Distributivité de la projection sur l'union (pareil pour la différence)
$\pi_{A1,\ldots,An}(R \cup S) \equiv \pi_{A1,\ldots,An}(R) \cup \pi_{A1,\ldots,An}(S)$

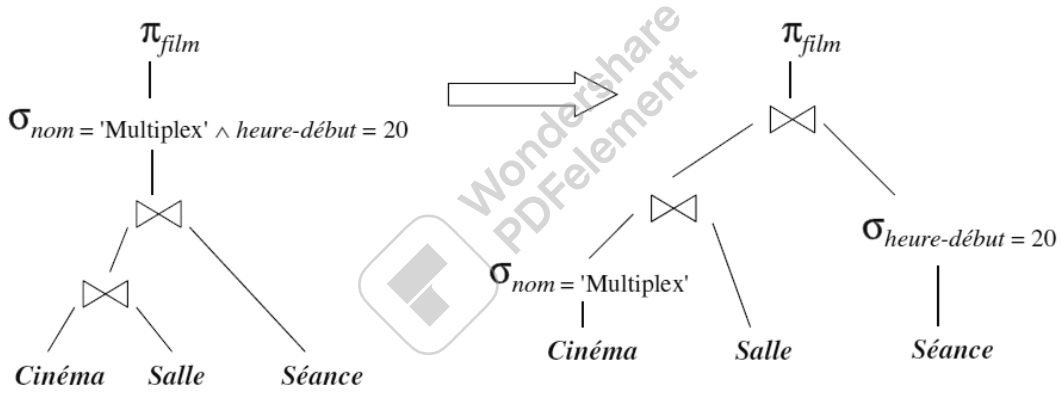
L'objectif principale de l'optimiseur est d'appliquer ces règles pour restructurer (réécrire) la requête afin de réduire le coût. Un algorithme de restructuration simple peut être donné par :

1. Séparer les sélections à plusieurs prédicats en plusieurs sélections à un prédicat (règle de regroupement des sélections)
2. Descendre les sélections le plus bas possible dans l'arbre (règles de commutativité et distributivité de la sélection)
3. Regrouper les sélections sur une même relation (règle de regroupement des sélections)
4. Réaliser les sélections d'abord, pour réduire la taille des données
   - On réalise d'abord les sélections, car c'est l'opérateur le plus "réducteur"
   - On réalise les jointures (opération très coûteuse) une fois que la taille des données a été réduite au maximum

La restructuration de la requête des deux exemples précèdent est donnée par :

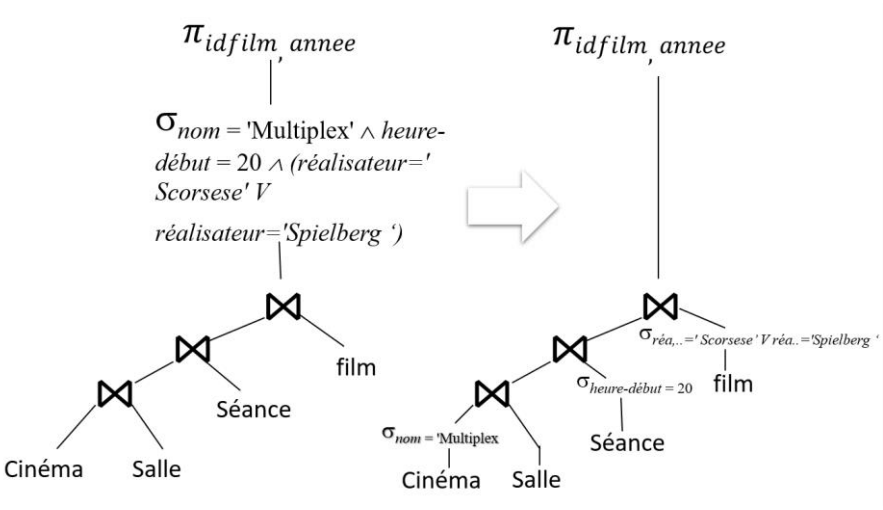
Initiale: π_{film} ($\sigma_{nom = \text{'Multiplex'} \wedge heure\text{-}début = 20}$ ((*Cinéma* ⋈ *Salle*) ⋈ *Séance*)

Optimisé: π_{film} (($\sigma_{nom = \text{'Multiplex'}}$ (*Cinéma*) ⋈ *Salle*) ⋈ $\sigma_{heuredébut = 20}$ (*Séance*))



Pour l'exemple-2 :
Initiale : $\pi_{idfilm,\ annee}$ ($\sigma_{nom = \text{'Multiplex'} \wedge heure\text{-}début = 20 \wedge (réalisateur=\text{' Scorsese' V } réalisateur=\text{'Spielberg '})}$ (((*Cinéma* ⋈ *Salle*) ⋈ *Séance)* ⋈ *film*)

Optimisé : $\pi_{idfilm,\ annee}$ ((($\sigma_{nom = \text{'Multiplex'}}$(*Cinéma)* ⋈ *Salle)* ⋈$\sigma_{heure\text{-}début = 20}$(Séance)) ⋈ $\sigma_{réalisateur=\text{' Scorsese' V } réalisateur=\text{'Spielberg '}}$(film))

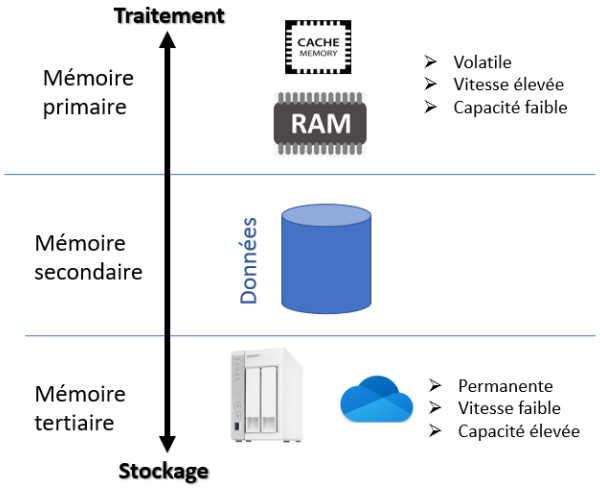


L'optimisation par réécriture algébrique est nécessaire mais pas suffisante. Il faut prendre en considération d'autres critères comme le mode d'accès aux données, la complexité de l'algorithme utilisés pour réaliser la jointure et les propriétés statiques de la BDD.

III- L'Optimisation : Le mode d'accès aux données

III-1- L'organisation physique des données

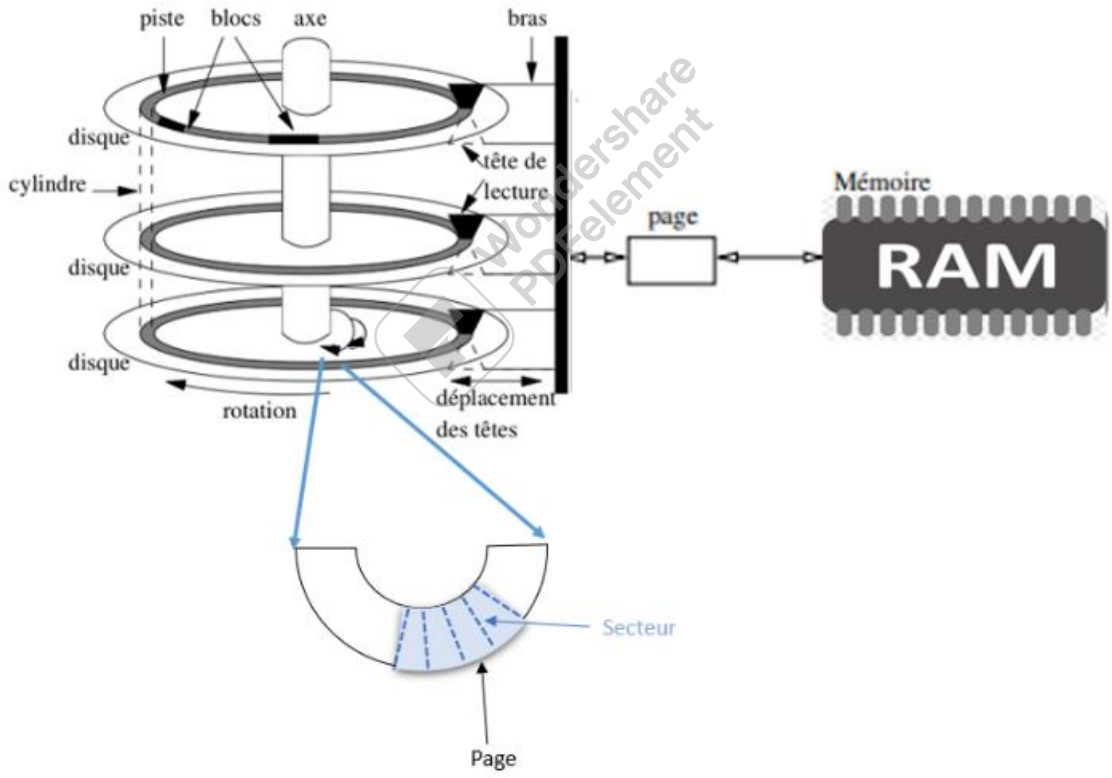
Une base de données est constituée, matériellement, d'un ou plusieurs fichiers stockés sur un support non volatile. Le support le plus couramment employé est le disque dur qui présente un bon compromis en termes de capacité de stockage, de prix et de performance. Les disque SSD (Solid State Drive), dont les performances sont nettement supérieures mais à des prix élevés, sont de plus en plus utilisé. Au cours des traitements, les données concernées sont chargées dans les mémoires principales volatiles et très rapides mais de taille très limitée : mémoire vive et cache. Les données traitées peuvent être archivées dans des mémoires tertiaires permanentes de grande capacité de stockage mais de faible vitesse.



La mémoire secondaire est divisée en pages (ou blocs) de taille égale constituées de même nombre de secteurs[1] contigus. La page est l'unité d'échange entre les mémoires secondaire et principale. Ainsi le coût des opérations dans les BDD est évalué principalement par le nombre de lectures/écritures de pages c.-à-d. accès au disque.
Les données sur disque sont stockées dans des fichiers. Chaque fichier occupe plusieurs pages sur disque et l'accès est géré par le système de gestion de fichiers du système d'exploitation. Un fichier est identifié par son nom.
Un fichier stocke un ensemble d'articles (enregistrements, n-uplets, lignes). Un article est une séquence de champs ou attributs. Il existe deux types d'articles :

- Articles en format fixe : la taille de chaque champ est fixée
- Articles en format variable : la taille de certains champs est variable.

Les articles sont stockés dans des pages, en général taille article < taille page, et l'adresse d'un article (ROWID) est composé de l'adresse page plus l'indice de l'article dans la page.



Les Blocs Oracle

La taille d'un bloc peut être choisie au moment de l'initialisation d'une base, et correspond obligatoirement à un multiple de la taille des blocs du système d'exploitation. À titre d'exemple, un bloc dans un système comme Linux occupe 1024 octets, et un bloc ORACLE occupe typiquement 4 096 ou 8 092 octets.

[1] Un secteur représente la plus petite surface d'adressage

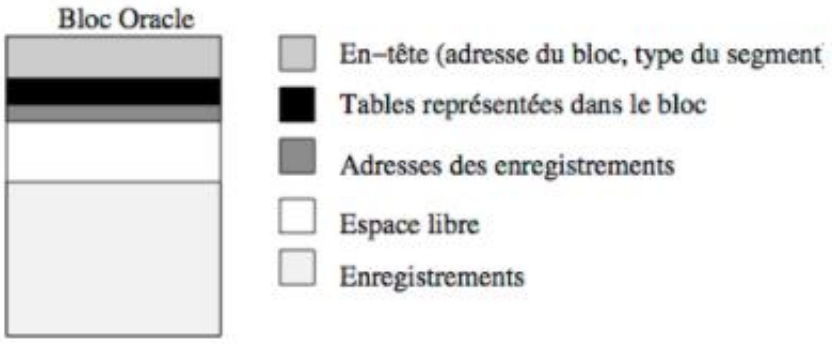


Fig. Structure d'un bloc Oracle

La structure d'un bloc est identique quel que soit le type d'information qui y est stocké. Elle est constituée des cinq parties suivantes :

- *L'entête* (*header*) contient l'adresse du bloc, et son type (données, index, etc.);
- Le *répertoire des tables* donne la liste des tables pour lesquelles des informations sont stockées dans le bloc ;
- Le *répertoire des enregistrements* contient les adresses des enregistrements du bloc ;
- Un *espace libre* est laissé pour faciliter l'insertion de nouveaux enregistrements, ou l'agrandissement des enregistrements du bloc (par exemple un attribut à NULL auquel on donne une valeur par un update).
- Enfin, *l'espace des données* contient les enregistrements.

Les trois premières parties constituent un espace de stockage qui n'est pas directement dédié aux données (Oracle le nomme *l'overhead*). Cet espace « perdu » occupe environ 100 octets. Le reste permet de stocker les données des enregistrements.

III-2- Accès aux données

Selon leur organisation physique, on peut accéder aux données d'une table par accès séquentiel, par index ou par hachage.

a- Accès Séquentiel :

Données non triées : Dans le cas d'une sélection à un au plusieurs critères On doit, partir du début du fichier, charger un par un tous les enregistrements en mémoire centrale, et effectuer à ce moment-là le test sur les critères de sélection. Ceci augmente linéairement le coût de la recherche en fonction du nombre d'enregistrement ([2]Complexité $O(n)$).

Données triées : Si le tableau est trié sur le champ servant de critère de recherche, il est possible d'effectuer une recherche par dichotomie qui est beaucoup plus rapide. Cet algorithme permet de diminuer par deux, à chaque étape, la taille de l'espace de recherche. Si cette taille, initialement, est de n blocs, elle passe à $n/2$ à l'étape 1, à $n/2^2$ à l'étape 2, et plus généralement à $n/2^k$ à l'étape k. Au pire des cas, la recherche se termine quand il n'y a plus qu'un seul bloc à explorer, autrement dit quand k est tel que $n \approx 2^k$ ce qui implique $k \approx \log_2(n)$ (Complexité

[2] Le comportement asymptotique du temps d'exécution, lorsque la taille des entrées n tend vers l'infini, et l'on utilise couramment les notations grand O de Landau.

O(log(n)). Ce pendant le maintien de l'ordre dans un fichier soumis à des écritures (insertions, suppressions et mises à jour) est très difficile à obtenir. Le principe de tri pour effectuer des recherches efficaces est à la source de très nombreuses structures d'index.

b- Accès Séquentiel Indexé :

Un index est une structure de données utilisée par les systèmes d'information pour lui faciliter de retrouver rapidement des données. L'utilisation des index dans une base de données facilite et accélère les opérations de recherche de tri, de jointure ou d'agrégation effectuées par le SGBD.

Oracle créé systématiquement un index chaque fois que l'on crée une clef primaire (PRIMARY KEY) ou une contrainte d'unicité (UNIQUE) sur une table. Les index doivent être utilisés sur les tables qui sont fréquemment soumises à des recherches.
Les index doivent être utilisés sur les attributs qui sont :

- Souvent mobilisés dans une jointure (clé Etrangère)
- Très discriminés (c'est à dire pour lesquels peu d'enregistrements ont les mêmes valeurs)
- Rarement modifiés

Inconvénients des index

- Les index diminuent les performances en mise à jour (puisqu'il faut mettre à jour les index en même temps que les données).
- Les index ajoutent du volume à la base de données et leur volume peut devenir non négligeable.

L'indexation des fichiers

Les SGBD créé les index en choisissant dans une table un ou plusieurs attributs, dont les valeurs constituent la *clé d'indexation* (par analogie à un dictionnaire c'est le premier mot d'une page). On associe à chaque valeur la liste des *adresse(s)* vers le (ou les enregistrements) correspondant à cette valeur (c'est l'équivalent des numéros de page dans un dictionnaire). Et finalement, on trie cette liste selon l'ordre alphanumérique pour obtenir l'index.
La signification des termes propres à l'indexation.

- Une **clé** (d'indexation) est la liste (l'ordre est important) d'attributs indexés
- Une **adresse** est un emplacement physique dans la base de données, qui peut être soit celle d'un bloc (ensemble d'enregistrement charger dans la RAM), soit un peu plus précisément celle d'un enregistrement (rowid) dans un bloc.
- Une **entrée** (d'index) est un enregistrement constitué d'une paire de valeurs (clé-adresse).

Un **index** est donc un fichier structuré dont les enregistrements sont des *entrées*.

i- *Index non dense :*

C'est le cas où le fichier est trié sur la clé d'indexation. La figure ci-dessous montre un index sur la table film de 16 tuples, la clé étant le titre du film, et les « A_i » symbolisent les adresses. On suppose que chaque bloc du fichier de données (contenant les films) contient 4 enregistrements, ce qui donne un minimum de quatre blocs. Il suffit alors de quatre entrées *[titre, adr]* pour indexer le fichier. Les titres utilisés sont ceux des premiers enregistrements de chaque bloc.
On remarque que les valeurs de clé existant dans le fichier de données ne sont pas toutes représentées dans l'index : on dit que l'index est *non-dense*. Puisque le fichier est trié sur la clé on ne fait figurer dans l'index que les valeurs de clé du *premier* enregistrement de chaque bloc.

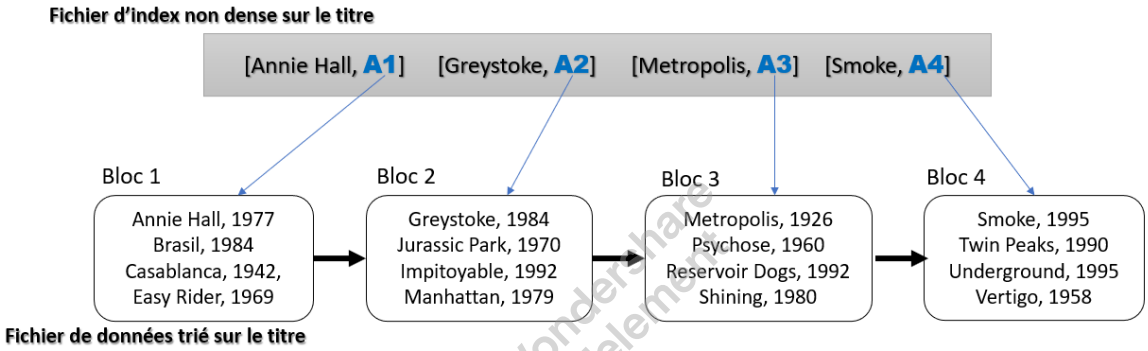


Figure : Index non dense

Supposons que l'on recherche le film *Shining*. En consultant l'index on constate que ce titre est compris entre *Metropolis* et *Smoke*. On en déduit donc que *Shining* se trouve dans le même bloc que *Metropolis* dont l'adresse est donnée par l'index. Il suffit de lire ce bloc et d'y rechercher l'enregistrement.
Le coût d'une recherche dans l'index est considérablement plus réduit que celui d'une recherche dans le fichier principal. D'une part les enregistrements dans l'index (les entrées) sont beaucoup plus petits que ceux du fichier de données puisque seule la clé (et une adresse) y figure. D'autre part l'index ne comprend qu'un enregistrement par bloc.

ii- *Index dense :*

C'est le cas où le fichier est trié sur un attribut différent de la clé d'indexation. Cela aura deux conséquences d'une part toutes les valeurs de l'attribut choisi pour créer l'index seront reportées dans l'index, ce qui accroît considérablement la taille de ce dernier, et d'autre part qu'à une même valeur de la clé d'indexation sont associées plusieurs adresses correspondant aux enregistrements liés à cette valeur de la clé (l'index n'est pas *unique*).
La figure ci-dessous montre le même fichier contenant seize films, trié sur le titre, et indexé maintenant sur l'année de parution des films. On constate d'une part que toutes les années du fichier de données sont reportées dans l'index, ce qui accroît considérablement la taille de ce dernier, et d'autre part qu'à une même année sont associées plusieurs adresses correspondant aux films parus cette année-là (l'index n'est pas unique).

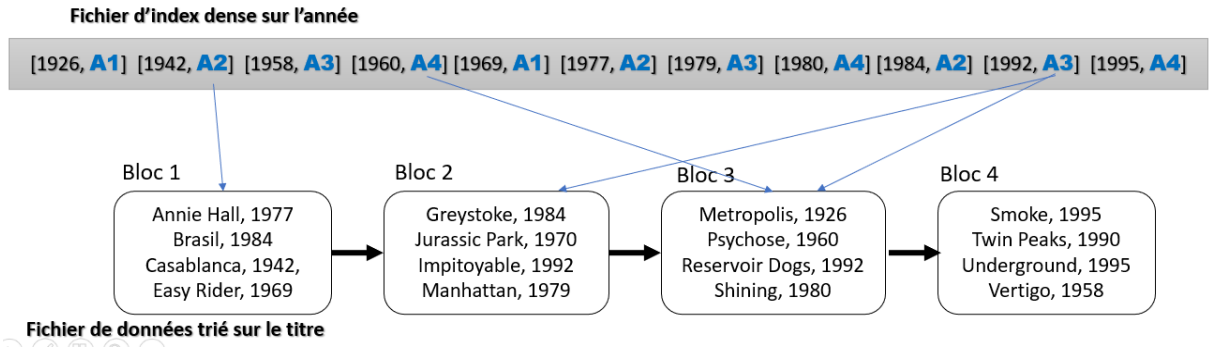


Figure : Index dense

La recherche par index dense est moins performante de celle dans un index non-dense. En effet si on cherche une valeur dans un fichier de données (par exemple les films d'une année) il faut lire dans l'index toutes les adresses correspondant au critère de recherche (tous les films de l'année), c.-à-d. qu'on peut lire dans plusieurs blocs. Ceci est illustré dans l'exemple précèdent où on a cherché dans deux blocs pour trouver les films parus en 1992.

Dans un fichier de données on ne peut créer qu'un seul index non dense, si on envisage de trier un fichier sur la clé primaire, par contre on peut créer plusieurs index denses pour les attributs faisant fréquemment de critère de recherche.

iii- Index multi-niveaux

Pour surmonter le problème de la taille des index dense, on a choisi d'indexer le fichier d'index lui-même. Puisque qu'un index est un fichier constitué d'entrées [clé, adr], trié sur la clé. *Ce tri nous permet d'utiliser, dès le deuxième niveau d'indexation, les principes des index non-denses*: on se contente d'indexer la première valeur de clé de chaque bloc.
Reprenons l'exemple de l'indexation des films sur l'année de parution. On peut alors créer un deuxième niveau d'index, comme illustré sur la figure ci-dessous. L'index de second niveau est construit sur la clé du premier enregistrement de chaque bloc de l'index de premier niveau. On diminue donc le nombre d'entrées.
L'intérêt d'un index multi-niveau est de pouvoir passer, dès le second niveau, d'une structure dense à une structure non-dense. Si ce n'était pas le cas on créer d'autre niveau jusqu'à ce que la granularité de lecture soit le bloc, on peut dire que quand un niveau d'index tient dans un seul bloc, il est inutile d'aller plus loin. Ceci est illustré dans la figure ci-dessous.

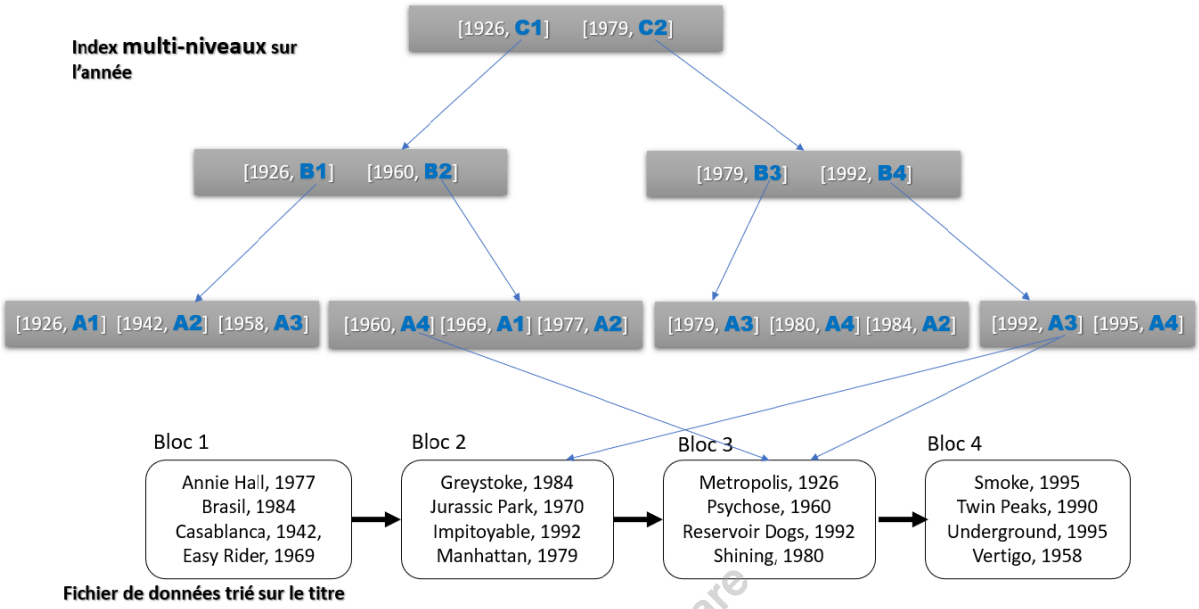


Figure : Index multi-niveaux

Les index multi-niveaux sont très efficaces en recherche, même pour des BDD de très grande taille. L'inconvenant est, comme toujours, la difficulté de maintenir des fichiers compacts et triés. L'arbre-B, étudié dans la section qui suit, donne des performances équivalentes à celles des index multi-niveaux et utilise des algorithmes de réorganisation dynamique qui résolvent la question de la maintenance d'une structure triée.

c- Accès séquentiel indexé avec arbres B

L'arbre-B est aussi une structure multi-niveau amélioré pour supporter les insertions et les suppressions et les recherche par intervalle. Elle est utilisée dans tous les SGBDR.

Si un bloc peut contenir au maximum *n entrées*, alors *l'ordre de l'a*rbre B est n/2. L'ordre d'un arbre dépend de la taille de la clé. Soit « c » la taille de la clé à indexer et « a » la taille d'une adresse et « b » l'espace utile d'un bloc, alors le nombre maximum d'entrées n est $n = \frac{b}{c+a}$.
Donc l'ordre de l'arbre est $\frac{b}{2(c+a)}$

Les arbres-B d'ordre[3] k sont :

- Des arbres équilibrés[4]
- Chaque nœud occupe une page[5] et contient *n* entrées tel que $k \leq n \leq 2k$. Seule exception : la racine qui contient entre 1 et 2k entrées
- Les entrés d'un nœud sont triés sur la clé
- Chaque nœud interne avec *n* entrées a n+1 fils

[3] L'ordre d'un arbre B correspond au nombre minimal d'entrées contenues dans chacun des blocs

[4] Est un arbre qui maintient une profondeur équilibrée entre ses branches. Cela a l'avantage que le nombre de pas pour accéder à la donnée d'une clé est en moyenne minimisé.

[5] L'unité d'échange entre les mémoires secondaire et principale

- Chaque nœud interne est un index pour ses fils/descendants
- Les n entrées sont triés suivant l'ordre croissant ($k \leq n \leq 2k$)
  - $C_1 \leq C_2 \leq \ldots \leq C_n$: clés
  - R_1 , …, R_n : information associée aux clés En général R_i = l'adresse physique (ROWID) de l'entrée de clé C_i
- n+1 pointeurs vers les nœuds fils dans l'index: P_1 , …, P_{n+1}
  - Contrainte : toutes les clés x dans le sous-arbre pointé par P_i respectent la condition $C_{i-1} < x < C_i$ ($2 \leq i \leq n$)
  - Pour i=1→ $x < C_1$, pour i=n+1 → $x > C_n$

- Le niveau le plus bas est appelé « niveau des feuilles », constitue un index *dense* sur le fichier de données. On trouve dans ce niveau les *entrées* d'index composées des pairs valeur de clé et adresse vers un enregistrement du fichier de données

La figure ci-dessous montre un arbre-B indexant notre collection de 16 films, avec pour clé d'indexation l'année de parution du film.

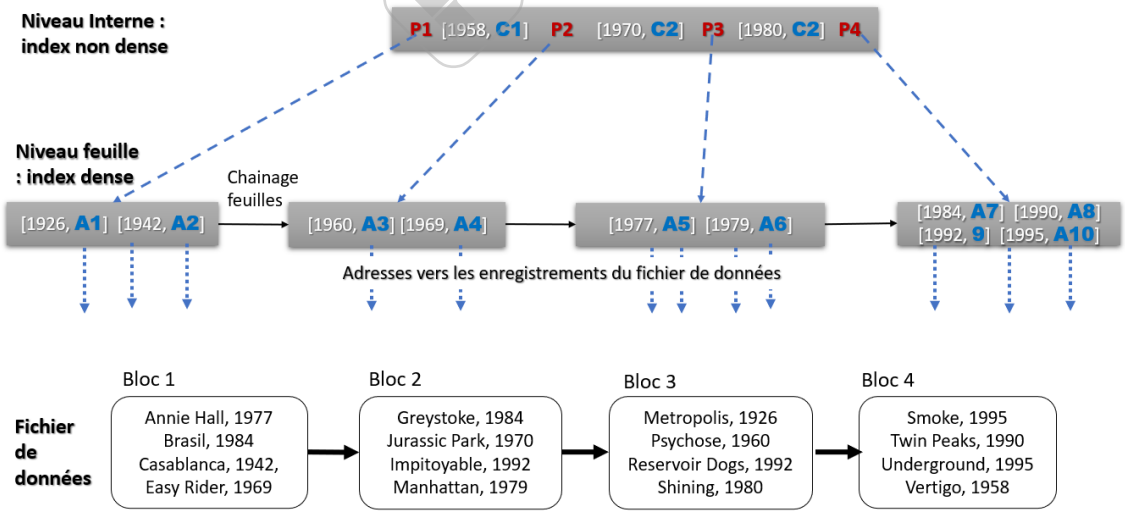


Figure : L'index -B sur l'année de parution des films

Contrairement aux fichiers d'index traditionnels, le niveau des feuilles de l'arbre B n'est pas constitué d'une séquence contiguë de blocs, mais d'une *liste chaînée* de blocs, ce qui permet de parcourir ce niveau, en suivant le chaînage. Cette organisation est moins efficace qu'un stockage continu (le passage d'un bloc à un autre peut entraîner un déplacement des têtes de

lecture), mais permet la réorganisation rapide et dynamique de la liste pour y insérer de nouveaux blocs.

Les niveaux de l'arbre B situés au-dessus des feuilles (sur notre exemple il n'y en a qu'un) sont les « niveaux *internes* », constituent un index non-*dense* sur le « niveau des feuilles ». Ils sont eux aussi constitués de blocs indépendants, mais sans chaînage associant les blocs d'un même niveau. Chaque bloc interne sert d'index local pour se diriger, de bas en haut, dans la structure de l'arbre, vers la feuille contenant les valeurs de clé recherchées.

Recherche d'un nœud dans un arbre-B

La recherche est effectuée de la même manière que dans un arbre binaire de recherche. Partant de la racine, on parcourt récursivement l'arbre ; à chaque nœud, on choisit le sous-arbre fils dont les clés sont comprises entre les mêmes bornes que celles de la clé recherchée. La difference avec l'arbre binaire c'est que le nœud dans un arbre-B est composé de plusieurs clés.

On définit la structure nœud composée des membres :

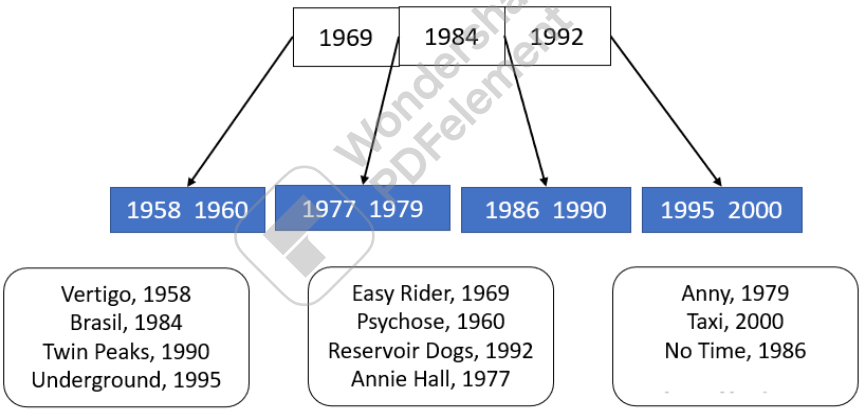
- taille : le nombre de clés dans chaque nœud
- clé[] : tableau des clés d'un nœud
- branche[] ; tableau de pointeur vers les arbres fils.
- feuil : variable Boolean informant si un nœud appartient au niveau feuil ou non.

L'algorithme de Recherche dans un arbre-B qui permet de trouver un clé x dans un nœud est donnée par :

```
fonction Recherche(nœud, x)
i=0
tant que i<nœud.taille et x>nœud.clé[i]
        i=i+1
si nœud.feuil
        si x=nœud.clé[i]
                renvoyer clé[i]
        sinon si i=nœud.taille
                renvoyer « Valeur non trouvée »
        FinSi
sinon si i=nœud.taille et x>nœud.cle[i]
        Recherche(nœud.branche[i-1], x)
sinon
        Recherche(nœud.branche[i], x)
FinSi
```

L'insertion dans un arbre-B

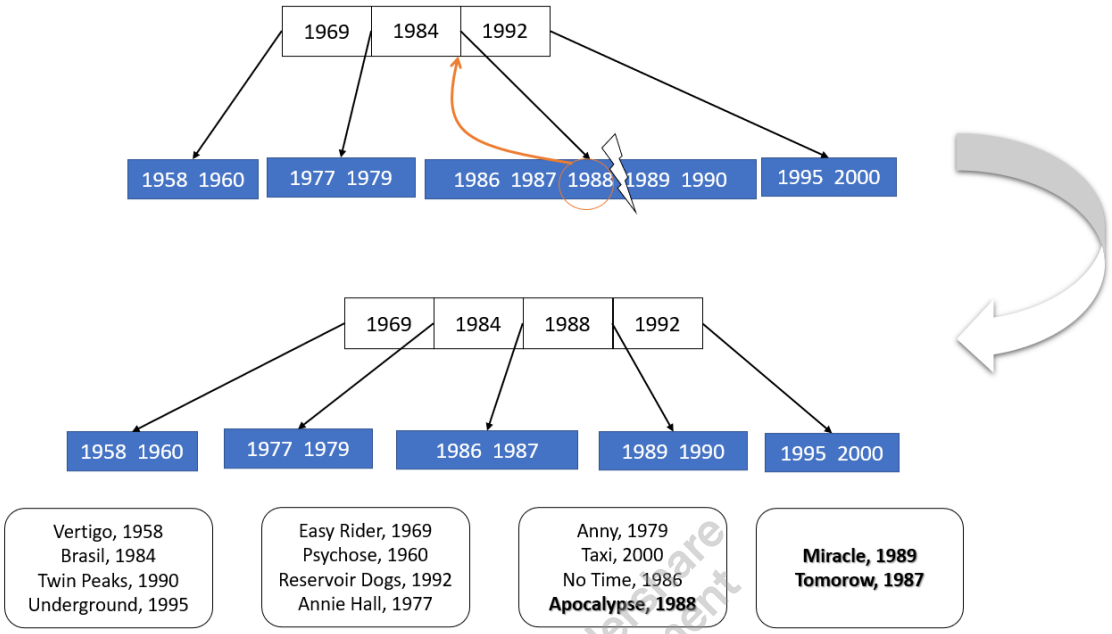
Nous allons voir comment le système maintient un arbre-B sur un fichier (une table) sans organisation particulière : les enregistrements sont placés les uns après les autres dans l'ordre de leur insertion, dans une structure dite séquentielle.
Comme exemple nous construisons un arbre B sur le titre des films. Nous avons donc deux fichiers : la table et l'index. Le premier a une structure séquentielle et contient les enregistrements représentant les lignes de la table. Pour faciliter la présentation on suppose que l'on peut mettre 4 enregistrements par bloc. Le second fichier a une structure d'arbre B, et ses blocs contiennent les entrées de l'index. On va supposer que l'on met aux plus quatre entrées par bloc (ordre de l'arbre est 2).
Initialement on suppose qu'on a onze enregistrements à indexé sur l'année de parution. Le principe de la construction de l'arbre est que lorsqu'un bloc dépasse les *2k* entrés on divise le bloc en deux et on fait monter la clé medium au niveau supérieur. L'arbre est pour l'instant constitué d'un niveau feuille constitué de quatre blocs et d'un niveau interne. Pour simplifier la schématisation nous présentons dans le niveau feuille que les clés d'indexation et dans les feuilles internes les pointeurs sont remplacés par les bordures internes.



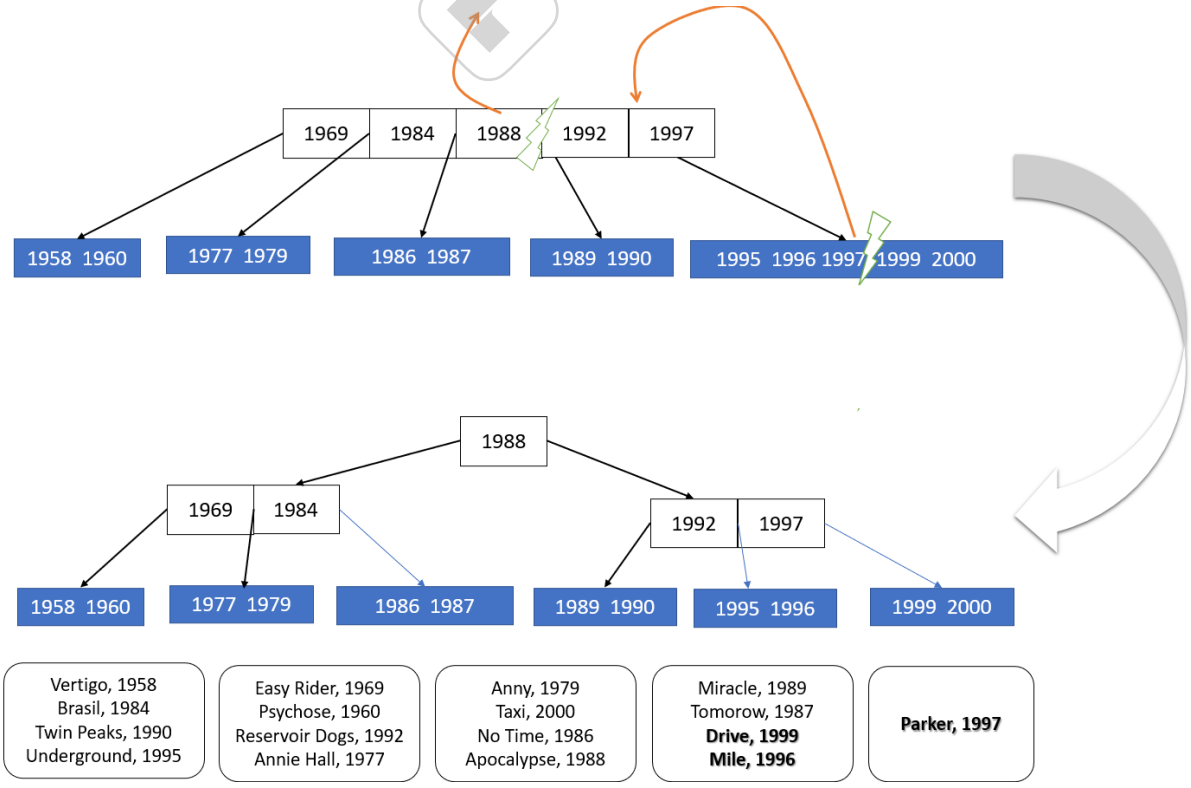
D'une part on insère le nouveau tuple dans la table d'une manière séquentielles, c.-à-d. l'insertion se fait après la dernière ligne de la table et dans le dernier bloc. Et d'autre part on insère une nouvelle entrée dans l'index en procédant de la manière suivante :

- On cherche l'endroit où on doit insérer la clé d'indexation (année). On part de la racine, et on suit les adresses comme si on recherchait dans un arbre de recherche un enregistrement pour la valeur de clé à insérer.
- Si la feuille où on veut insérer n'est pas pleine (<2k) on ajoute la nouvelle entée sans porter aucun changement aux nœuds des feuilles internes mais en maintenant les valeurs triées.
- Si non il y a éclatement du nœud et remontée d'une manière récursive de la clé médiane au niveau du père.

La figure suivante montre la réorganisation de l'arbre après l'insertion de trois films des années suivant l'ordre de l'insertion 1988, 1989 et 1987. Nous constatons que l'arbre à toujours 2 niveaux mais avec une racine plaine.



On continue la saisie dans la table films en insérant cette fois trois films des années 1999, 1996 et 1997. On constate que ses insertions ont provoqué l'éclatement de la racine et donc l'ajout d'un autre niveau.

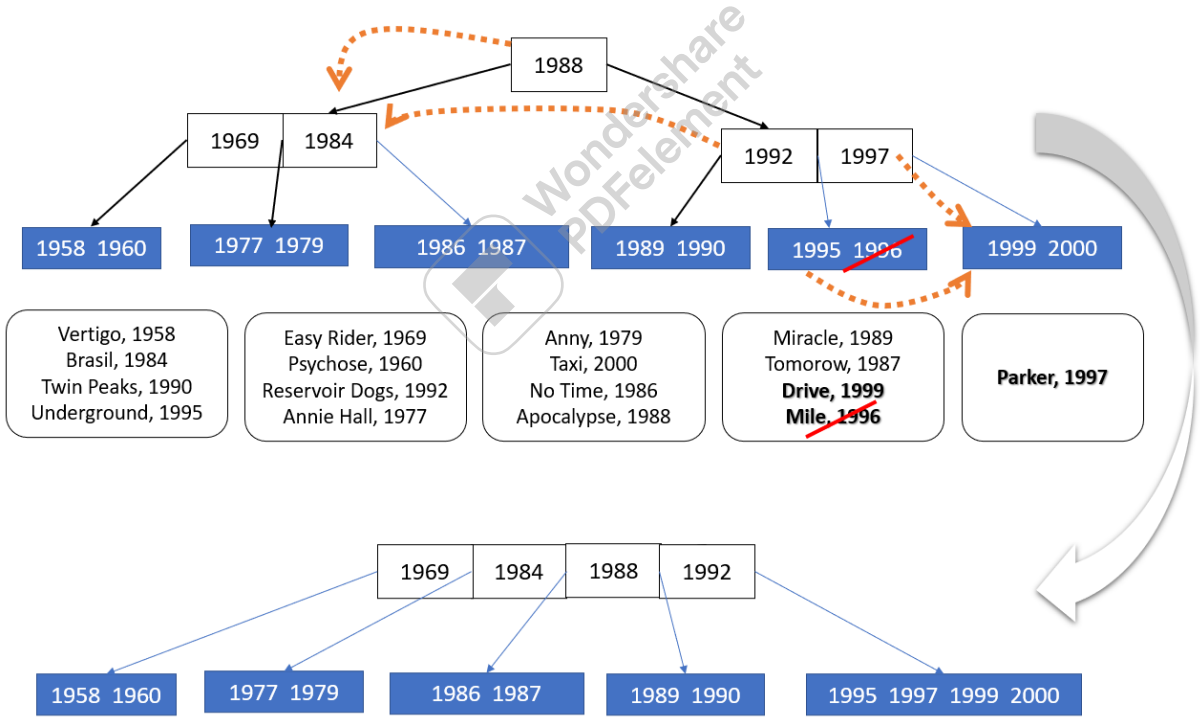


La suppression dans un arbre-B

La suppression se fait toujours au niveau des feuilles

- Si la clé à supprimé n'est pas dans une feuille alors on la remplace par la plus grande valeur des plus petite des clés des fils (ou bien par la plus petite des plus grand)
- Si la suppression de la clé d'une feuille (récursivement d'un nœud) amène à avoir moins de k clés :
  - Combinaison avec un nœud voisin (avant ou après)
  - Descente de la clé associant ces deux nœuds (éclatement du nœud si nécessaire et donc remonté d'une nouvelle clé)

Prenant l'exemple de la figure précédente, nous allons supprimer du fichier le films *Mile, 1996.* Lorsqu'on va supprimer la clé 1996 on va avoir une feuille d'une seule valeur. On fusionne donc 1995 dans la feuille de droite et on fait descendre la clé 1997. Cette descente va laisser dans le nveau interne un nœud à une seule valeur 1992 (<k) donc on va le fusionner avec celui du gauche et on fait descendre la clé 1988 (traitement récursive).



d- Accès séquentiel indexé avec le Hachage

Le hachage est utilisé par les SGBD pour organiser de grandes collections de données sur mémoire permanente. C'est une structure de donnée de hachage créé en partie dans la mémoire principale RAM. Nous étudions deux types de hachage : le *hachage statique* ne fonctionne correctement que pour des collections de tailles fixes, ce qui exclut des tables évolutives (le cas le plus courant). Le *hachage dynamique*, qui s'adapte à la taille de la collection indexée.

Hachage statique

Indexé par hachage une table suivant une clé « c » revient à utiliser une fonction (dite *de hachage*) qui, pour chaque valeur de clé c, donne l'adresse f(c) d'un espace de stockage où l'élément doit être placé. En mémoire principale cet espace de stockage est en général une liste chaînée, et en mémoire secondaire une séquence de blocs sur le disque que nous appellerons *fragment* (*bucket* en anglais).

Chaque fragment est numéroté et a une adresse. Cette dernière est celle de son premier bloc. Une table de hachage permet d'associer le numéro du fragment à son adresse. C'est donc un tableau à deux dimensions, avec les numéros dans une colonne et les adresses dans une autre.

Prenons l'exemple de notre ensemble de films, et organisons-le avec une table de hachage sur le titre. Pour simplifier, on va supposer que chaque fragment contient un seul bloc avec une capacité d'au plus quatre films. L'ensemble des 16 films occupe donc au moins 4 blocs. Pour garder un peu de souplesse dans la répartition qui n'est pas toujours uniforme, on va affecter 5 fragments à la collection de films, et on numérote ces fragments de 0 à 4.

Pour affecter un film à l'un des fragments. On utilise une fonction qui, appliquée à un titre, va donner en sortie un numéro de fragment. Le résultat de la fonction, pour n'importe quelle chaîne de caractères, doit être un entier compris entre 0 et 4

Nous pouvons utiliser un principe simple pour notre exemple, en considérant la première lettre du titre, et en lui affectant son rang dans l'alphabet. Ensuite, pour se ramener à une valeur entre 0 et 4, on prendra simplement le reste de la division du rang de la lettre par 5 (« modulo 5 »). La fonction h sera définie par :

$$h(titre) = rang(titre[0]) \bmod 5$$

Titre	Année
Juassic Park	2014
Annie Hall	2015
Brasil	2000
Manhattan	2001
Impitoyable	2008
Twin Peaks	1999
Underground	1989
Esay Rider	1977
Psychose	2020
Vertigo	2000
Greystoke	1988
Metropolis	1978
Shining	2001
Smoke	2003
Casablanca	2004
Reservoir	2018

0	Juassic Park	2014
	Twin Peaks	1999
	Esay Rider	1977
1	Annie Hall	2015
	Underground	1989
	Psychose	2020
2	Brasil	2000
	Vertigo	2000
	Greystoke	1988
3	Manhattan	2001
	Metropolis	1978
	Casablanca	2004
	Reservoir	2018
4	Impitoyable	2008
	Shining	2001
	Smoke	2003

Figure : Exemple de Table de Hachage

La figure ci-dessous montre la table de hachage obtenue avec cette fonction. Tous les films commençant par *a, f, k, p, u* et *z* sont affectés au bloc *f1*. Les lettres commençant par b, *g, l, q* et *v* sont affectées au bloc *f2* et ainsi de suite. Notez que la lettre '*e'* a pour rang 5 et se trouve donc affectée au bloc 0.

La distribution des adresse générées par une fonction de hachage risque de ne pas être homogène. En effet une fonction de hachage mal conçus peut engendrer facilement une surcharge des blocs en affectant la même adresse à plusieurs valeurs de la clé (titre de film). Dans le pire des cas, une fonction de hachage affecte tous les enregistrements à la même adresse, et la structure dégénère vers un simple fichier séquentiel.

Recherche dans une table de hachage

Les index par hachage sont appliqués surtout dans les requêtes SQL qui demande une sélection avec une expression logique utilisant l'égalité (Recherche d'une valeur dans un attribut).

Exemple :

Nous voulons effectuer la recherche suivante sur l'attribut indexé « titre » :


```
select *
from Film
where titre = 'Impitoyable'
```

La function de Hachage appliqué sur la valeur `'Impitoyable'` donne 4. On peut donc charger le fragment 4 et y trouver le film *Impitoyable*. On a donc pu effectuer cette recherche en lisant un seul bloc, ce qui est optimal.

Le hachage ne permet pas d'optimiser les recherches par intervalle, puisque l'organisation des enregistrements ne s'appuie pas sur le tri des clés. La requête suivante par exemple ne peut être résolue que par le parcours de tous les blocs de la structure, même si trois films seulement sont concernés.

```
select *
from  Film
where titre between 'Annie Hall' and 'Easy Rider'
```

Mises à jour

Le hachage statique offre des performances pour les recherches par clé, il est – du moins mal adapté aux mises à jour. Prenons tout d'abord le cas des insertions : comme on a évoqué avant les insertions conduisent à dépasser la taille estimée initialement pour les blocs. La seule solution est alors de chaîner de nouveaux fragments.

Cette situation est illustrée dans la figure ci-dessous. On a inséré deux nouveaux films, *Citizen Kane et Miracle*. La valeur donnée par la fonction de hachage est 3.

Insert into film values (Citizen Kane, 2010)
Insert into film values (Miracle, 1987)

h('Citizen Kane')=3
h(Miracle)=3

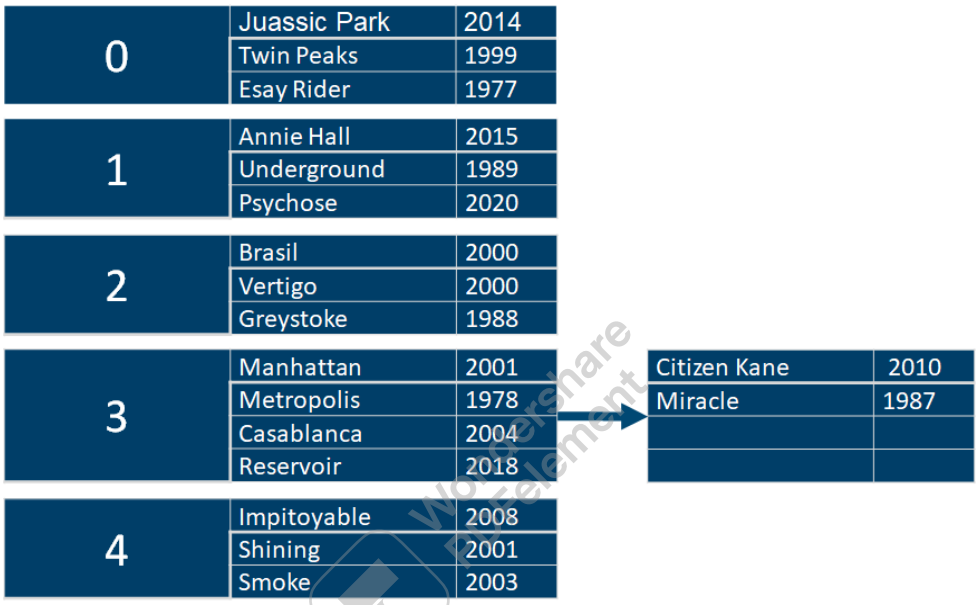


Figure- Table Hachage avec débordement

On doit donc stocker le film dans l'espace associé à la valeur 3 car c'est là que les recherches iront s'effectuer. On doit alors chaîner un nouveau fragment au fragment 3 et y stocker le nouveau film. Il y a donc une dégradation potentielle des performances puisqu'il faudra, lors d'une recherche, suivre le chaînage et inspecter tous les enregistrements pour lesquels la fonction de hachage donne la valeur 3.

Hachage dynamique

Le problème de débordement des blocs lié au hachage statique est le débordement de compartiment sera surmonté par le hachage dynamique appeler aussi **hachage extensible**. Dans cette méthode, la table de hachage augmente et diminue en fonction du nombre d'enregistrements. Il permet d'effectuer des opérations telles que l'insertion, la suppression, etc. sans affecter les performances.
Dans le Hachage dynamique :

- La fonction de hachage va retourner des bits (un nombre binaire)
  - $H(c)=b_0b_1b_2\ldots\ldots b_{n-1}$
- La table de Hachage et de taille variable. Elle varie de 2^n (1, 2, 4, 8, …)
- Quand un bloc déborde on l'éclate en utilisant la bit de hachage suivant.

Comme Exemple nous prenons toujours la table film et on suppose une fonction de hachage qui Hache le titre d'un film sur 1 octet. Nous obtenons le tableur de hachage suivant :

Titre	*h(titre)*
Vertigo	01110010
Brazil	10100101
Twin Peaks	11001011
Underground	01001001
Easy Rider	00100110
Psychose	01110011
Greystoke	10111001
Shining	11010011
Tomorow	01110110
Taxi	10110011
Anny	01110001

Nous supposons toujours que les blocs ne peuvent contenir que 4 n-uplet. Dans un premier temps il y a 8 n-uplet stockés donc nous nous intéressons seulement qu'au premier 0 ou 1. La figure ci-dessous montre l'insertion des six premiers films de notre liste, et leur affectation à l'un des deux blocs. Le film *Vertigo* par exemple a pour valeur de hachage 01110010 qui commence par 0, et se trouve donc affecté à la première entrée.

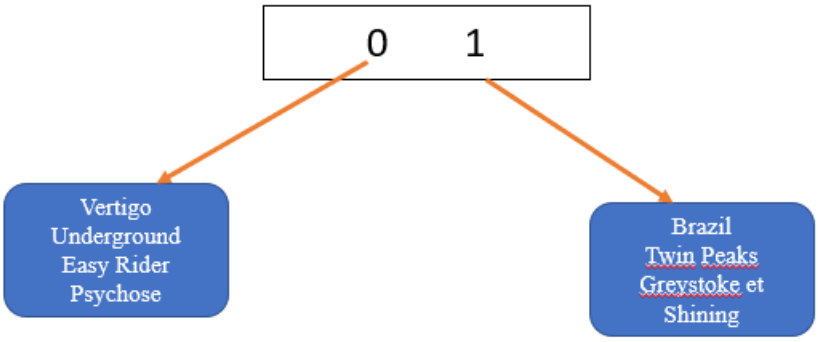


Figure : Hachage extensible avec 2 entrées

Nous effectuons l'insertion de *Tomorow*, avec pour valeur de hachage 01110110, entraine le débordement du fragment associé à l'entrée 0.
On va alors doubler la taille du répertoire pour la faire passer à quatre entrées, avec pour valeurs respectives 00, 01, 10, 11, soit les 2^2 combinaisons possibles de 0 et de 1 sur deux bits. Ce doublement de taille du répertoire entraine la réorganisation suivante :

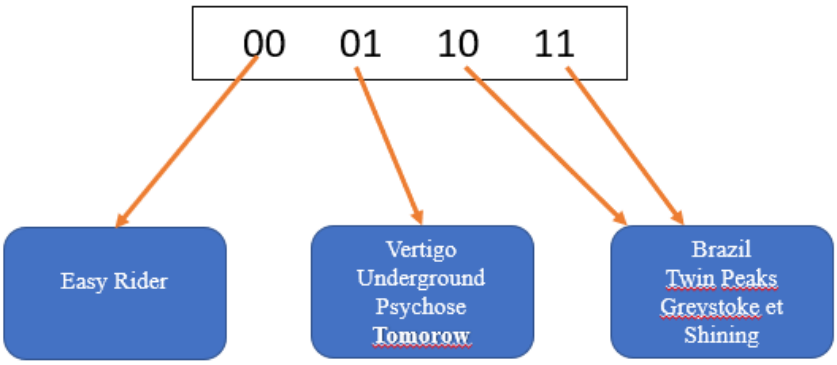


Fig. Doublement du TH dans le hachage extensible

On insère maintenant « Taxi ». Dans ce cas il aura éclatement du troisième fragment et création d'un nouveau sans doublé le tableau de hachage, et on répartit les adresses du répertoire pour référencer les deux fragments

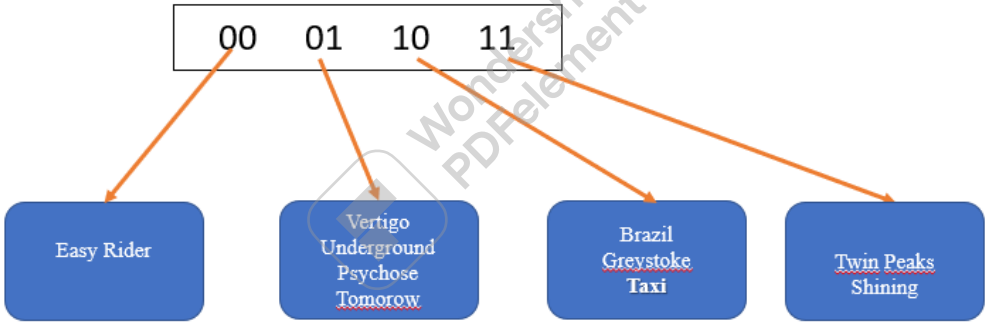
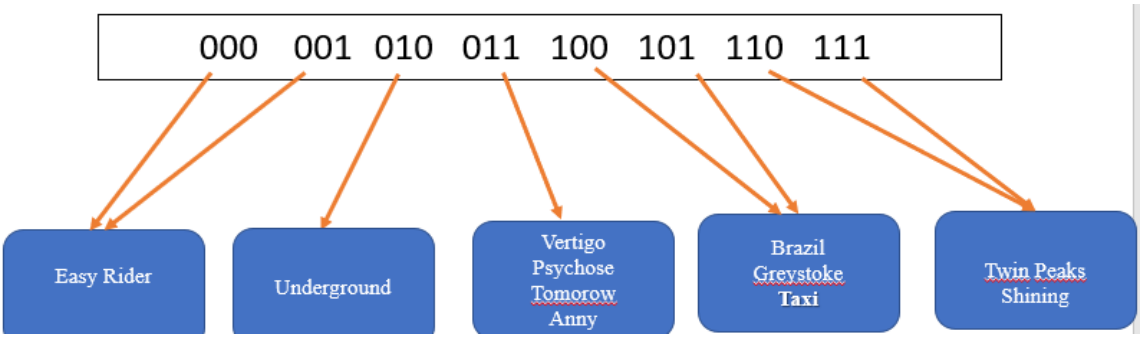


Fig. éclatement d'un bloc sans doublement du TH

On insère maintenant « Anny ». Il y aura donc éclatement du deuxième fragment en 010 et 011 avec doublement du TH.



En résumé, pour l'indexation par hachage il n'y a que deux cas possibles :

- **Cas 1** : Si on insère dans un fragment plein, mais plusieurs entrées pointent dessus. On alloue alors un nouveau fragment, et on répartit les adresses du répertoire pour référencer les deux fragments.
- **Cas 2** : Si on insère dans un fragment plein, associé à une seule entrée. On double à nouveau le nombre d'entrées.

IV- L'optimisation : Les algorithmes de jointure

La jointure est l'opération la plus couteuse le choix de l'algorithme qui va permettre la jointure entre deux ou plusieurs table est primordiale pour optimiser les requêtes. Plusieurs algorithmes peuvent être utilisés selon le mode d'accès aux données des attributs utilisés dans la jointure. Les principaux algorithmes sont :

- Boucles imbriquées simples
- Tri-fusion
- Jointure par hachage
- Boucles imbriquées avec index

IV-1 Jointure en boucles imbriquées (Nested Loop)

La jointure en boucles imbriquées est la plus ancienne est la plus basique. Dans cet algorithme le système accède aux lignes d'une table pilote ou externe en utilisant une boucle externe puis cherche, dans une table interne ou secondaire, les lignes qui correspondent aux critères de prédicat en utilisant une boucle interne. Une jointure par boucles imbriquées implique les étapes de base suivantes :

1. ***L'optimiseur détermine la source de ligne de pilotage et la désigne comme boucle externe***. La boucle externe produit un ensemble de lignes pour piloter la condition de jointure. La source de ligne peut être une table accessible à l'aide d'une analyse d'index (by index rowid), d'une analyse complète (Full) de la table ou de toute autre opération générant des lignes. Le nombre d'itérations de la boucle interne dépend du nombre de lignes récupérées dans la boucle externe.
2. ***L'optimiseur désigne l'autre source de ligne comme boucle interne.***

Le pseudo-code de cet algorithme est :

```
Algorithme boucles-imbriquées
Entrées: tables R, S
Sortie: table de jointure J
début
J = ⌽
pour chaque r dans R répéter
        pour chaque s dans S répéter
                si r joignable à s alors J := J U {r ⋈ s}
        fin répéter
fin répéter
fin
```

Calcul du coût

Le cout de cet algorithme est donné par le nombre totale de pages lus dans le disque. Lorsqu'on parcoure les n-uplet d'une table dans une boucle :

1. On lit sur disque les pages de la table pour les charger en mémoire
2. On lit en mémoire les articles de chaque page.

On suppose :

- Dans le Disque : Les n-uplets de R sont stockés dans N_R pages et ceux de S dans N_S pages
- Dans la Mémoire : 1 pages pour charger les n-uplets de R et 1 pages pour S.

Pour chaque page de R on lit toutes les pages de S et on fait la jointure page par page entre les articles. S est lue donc N_R fois et R une seule fois

Par conséquent le coût est : $C = N_R + N_R \times N_S$

C'est le nombre de page de R qui augmente le coût puisque. Il détermine le nombre de fois de lecture des pages de S. Si les n-uplet de R peuvent être charger dans K pages mémoire, le coût se réduit à : $C = N_R + \frac{N_R}{K} \times N_S$. Et si toutes la table R est charger en mémoire le coût devient : $C = N_R + N_S$

Conclusion :

La jointure par boucles imbriquées n'est efficace que le cas on a de petites tables pilotes et dans le cas où l'une des deux tables se charge en mémoire

Exemple1 :

Liste des salles du cinéma id=12

```
select * from cinema, salle
where cinema.idcinema=salle.idcinema
and cinema.idcinema = 12;
```

Le plan d'action est donné ci-dessous

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	3
NESTED LOOPS			1	3
TABLE ACCESS	CINEMA	BY INDEX ROWID	1	1
INDEX	SYS_C008484	UNIQUE SCAN	1	1
Access Predicates				
CINEMA.IDCINEMA=12				
TABLE ACCESS	SALLE	FULL	1	2
Filter Predicates				
SALLE.IDCINEMA=12				

La jointure est réalisée par boucle imbriquée. L'optimiseur accède à la table source « cinema » pour chercher le cinéma numéro 12. Cette accès se fait par le rowid trouvé par un accès unique

à l'index sur la clé primaire « cinema.idcinema ». Ensuite l'optimiseur utilise un accès complet à la table interne « salle » pour trouver les n-uplets correspondant au cinéma numéro 12.
Lorsqu'on crée l'index « indexidcinema » sur la clé étrangère « salle.idcinema », l'optimiseur accède à la table interne « salle » par lot de rowid. Ces derniers sont trouvés par chargement d'une rangé de ligne de l'index « indexidcinema » correspondant au numéro 12 (voir plan ci-dessous).

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	1
NESTED LOOPS			1	1
TABLE ACCESS	CINEMA	BY INDEX ROWID	1	1
INDEX	SYS_C008484	UNIQUE SCAN	1	1
Access Predicates				
CINEMA.IDCINEMA=12				
TABLE ACCESS	SALLE	BY INDEX ROWID BATCHED	1	0
INDEX	INDEXIDCINEMA	RANGE SCAN	1	0
Access Predicates				
SALLE.IDCINEMA=12				

Exemple-2 :
Afficher le cinéma contenant la salle numéro 124

```
select * from cinema, salle
where cinema.idcinema=salle.idcinema
and idsalle = 124 ;
```

Le plan d'action est donné ci-dessous

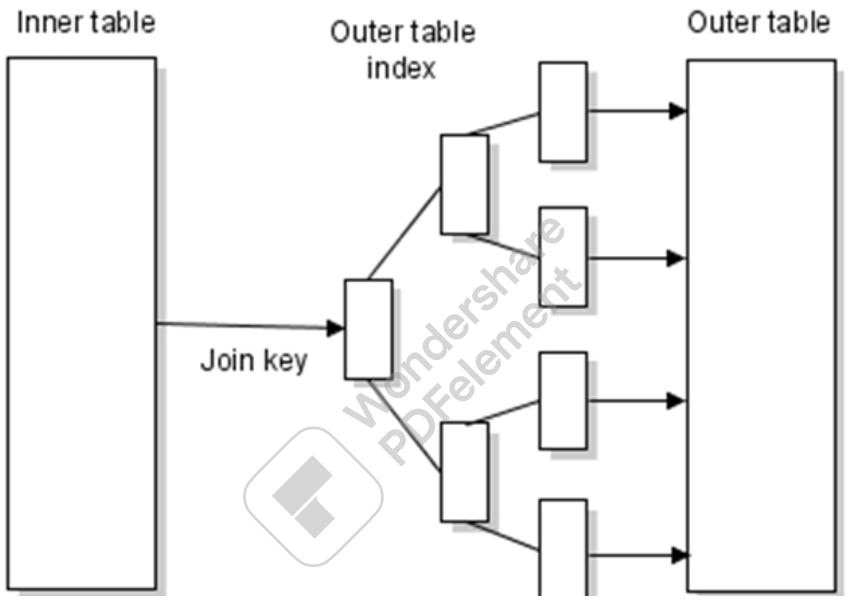
OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	2
NESTED LOOPS			1	2
TABLE ACCESS	SALLE	BY INDEX ROWID	1	1
INDEX	SYS_C008485	UNIQUE SCAN	1	1
Access Predicates				
IDSALLE=124				
TABLE ACCESS	CINEMA	BY INDEX ROWID	1	1
INDEX	SYS_C008484	UNIQUE SCAN	1	0
Access Predicates				
CINEMA.IDCINEMA=SALLE.IDCINEMA				

Cette fois l'optimiseur choisi la table salle comme source de ligne par un accès rowid trouvé par un accès à l'index sur la clé primaire de salle. Une fois la salle 124 est trouvée le système cherche le cinéma de cette salle en utilisant l'index sur la clé primaire de cinéma.

VI-2 Jointure par boucles imbriquées avec indexe

Ce le type de jointure de boucles imbriqué le plus courant puisque dans la plupart des cas on réalise, dans une requête SQL, une jointure entre une clé primaire qui est indexée par le SGBD et une clé étrangère. Donc on a au moins un index qui peut être utilisé pour réaliser la jointure rapidement.
Pour réaliser la jointure R.A=S.B

- Le système accède tous d'abord par un FullScan à la table S pour mémoriser la valeur de x=S.B de chaque n-uplet de B
- Le système cherche ensuite dans l'index de la clé primaire R.A en utilisant un Unique Scan pour trouver l'adresse où est stocké le n-uplet correspondant à cette clé.
- Le résultat de la jointure est donc une paire contenant le n-uplet S et une adresse de d'un n-uplet de R [S, adresseR].
- Connaissant maintenant l'adresse du n-uplet de R le système fait un accès direct à cet enregistrement en utilisant l'index rowid, et le résultat est [S, R].



Le pseudo-code de cet algorithme est :

```
Algorithme boucles-imbriquées
Entrées: tables R, S
Sortie: table de jointure J
début
J = ∅
pour chaque r dans R répéter
        pour chaque s dans Index_S.B(r.A) répéter
                si r joignable à s alors J := J U {r ⋈ s}
        fin répéter
fin répéter
fin
```

Exemple : soit le schéma suivant :

Film (IDfilm, réalisateur, année)
Séance (IDsalle, IDfilm, heuredébut)

Nous cherchons à connaitre les salles de chaque cinéma.

```
select * from cinema, salle
where cinema.idcinema=salle.idcinema;
```

le plan d'exécution est donné par la figure suivante :

NESTED LOOPS			1	3
NESTED LOOPS			1	3
STATISTICS COLLECTOR				
TABLE ACCESS	SALLE	FULL	1	2
INDEX	SYS_C008484	UNIQUE SCAN	1	0
Access Predicates				
CINEMA.IDCINEMA=SALLE.IDCINEMA				
TABLE ACCESS	CINEMA	BY INDEX ROWID	1	1

IV-3 Jointure par hachage (Hash Loop)

La jointure de hachage est le choix préférable des optimiseurs lorsque les tables de jointure sont de grande taille ou lorsque les jointures nécessitent la plupart des lignes de tables jointes. Ce type de jointure est valable seulement pour les jointures d'égalité.

L'optimiseur utilise la plus petite des deux tables de jointure pour construire une table de Hachage en mémoire, c'est la phase de construction. Une fois la table de hachage construite, l'optimiseur scanne la plus grande table. Pour chaque ligne de cette table, il recherche les lignes jointe de la relation de construction en consultant la table de hachage, c'est la phase de sonde.
Le seul but de la table de hachage est d'agir comme une structure temporaire en mémoire pour éviter d'utiliser un index pour accéder aux enregistrements de la table source (petite table). Une table de hachage est une structure de données non ordonnée qui peut mapper des clés à des valeurs. Ce mappage est réalisé par une fonction de hachage qui transforme chaque clé en un indice ou un code de hachage. Ce dernier permet un accès direct à l'enregistrement à travers la clé.
Lors de la construction de la table de hachage, si la taille de cette dernière dépasse la taille maximale de la mémoire, définie lors de la configuration de cette dernière, l'optimiseur interrompe cette construction, sonde l'autre table pour ajouter les tuples de jointure, Réinitialise la table de hachage puis continue à analyser l'entrée de construction de la petite table.

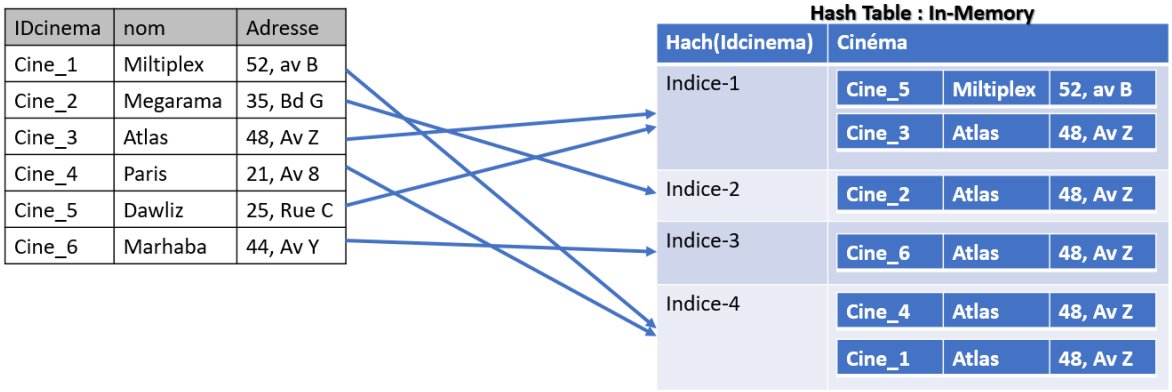


Figure : Construction d'une table de hachage en utilisant la fonction de hachage Hach(...)

L'algorithme qui gère la jointure de hachage, en prenant en considération la taille de la mémoire configurée pour stocker la table de Hachage, peut être donné par :

Entrées: tables R, S, TH
Sortie: table de jointure J (R.A = R.B)
procédure : inserer(TH, s, i) (insérer dans une tableau mémoire TH la structure s dans TH[i])
Fonction : FH (fonction de hachage)
taille_TH : taille table de hachage
taille_M : Taille mémoire configurer pour la hachage

```
début
Tantque (r dans R ET taille_TH<taille_M)
        inserer(TH, r, FH(R.A))
Fin Tantque
tantque taille_TH > taille_TM
        pour chaque s dans S répéter
                i :=FH(R.B)
                J := J U {TH(i) ⋈ s}
        fin répéter
        Tant que (r dans R ET taille_TH<TM) répéter
                inserer(TH, r, FH(R.A))
        Fin Tantque
Fin Tantque
Si r n'est plus dans R Alors
        pour chaque s dans S répéter
                i :=FH(R.B)
                J := J U {TH(i) ⋈ s}
        fin répéter
FIN Si
```

On peut décrire cet algorithme de la manière suivante :

Tant qu'on c'est la fin de R et la taille de la table de hachage est inférieur à la mémoire réservée.
Ajouter le tuple r à la table de Hachage
Tant que la taille de la table de hachage est égale à la mémoire réservée.

1. Scannez l'entrée de la table sonde et ajoutez des tuples de jointure
2. Réinitialiser la table de hachage et continuer à analyser l'entrée de construction

Si on arrive à la fin du tableau

1. Scannez l'entrée de la table sonde et ajoutez des tuples de jointure correspondante à la relation de sortie

Exemple :
Nous voulons afficher les cinémas dont le nom commence par « M » avec leurs salles.

```
SELECT * FROM cinema, salle
WHERE cinema.idcinema=salle.idcinema
AND cinema.nom like 'M%'
```

La figure ci-dessous montre les deux phases de la jointure par hachage : Etape de construction et la phase de sonde

IDcinema	nom	Adresse
Cine_1	Miltiplex	52, av B
Cine_2	Megarama	35, Bd G
Cine_3	Atlas	48, Av Z
Cine_4	Paris	21, Av 8
Cine_5	Dawliz	25, Rue C
Cine_6	Marhaba	44, Av Y

IDSalle	Capacite	IDcinema
S1	145	Cine_1
S2	544	Cine_2
S3	54	Cine_2
S4	541	Cine_1
S5	125	Cine_3
S6	123	Cine_4
S7	56	Cine_5
S8	147	Cine_5
S9	100	Cine_4
S10	120	Cine_6

WHERE nom like 'M%'

Phase Construction

Phase Sonde

Hash Table : In-Memory

Hach(Idcinema)	Cinéma		
Indice-1			
Indice-2	Cine_2	Atlas	48, Av Z
Indice-3	Cine_6	Atlas	48, Av Z
Indice-4	Cine_1	Atlas	48, Av Z

$\sigma_{nom\ like\ `M\%`}(Cinema) \bowtie Salle$

Cine_1	Atlas	48, Av Z	S1	145	Cine_1
Cine_2	Atlas	48, Av Z	S2	544	Cine_2
Cine_2	Atlas	48, Av Z	S3	54	Cine_2
Cine_1	Atlas	48, Av Z	S4	541	Cine_1
Cine_6	Atlas	48, Av Z	S10	120	Cine_6

Le plan d'exécution donné par l'optimiseur d'oracle est par la figure ci-dessous.

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			479	26
HASH JOIN			479	26
Access Predicates				
CINEMA.IDCINEMA=SALLE.IDCINEMA				
TABLE ACCESS	CINEMA	FULL	48	9
Filter Predicates				
NOM LIKE 'M%'				
TABLE ACCESS	SALLE	FULL	10000	17

Calcul du coût
On suppose que :

- Dans le Disque : Les n-uplets de R sont stockés dans N_R pages et ceux de S dans N_S pages
- Dans la Mémoire : K pages pour charger les n-uplets de R pour construire la table de hachage.

Par conséquent le coût est : $C = N_R + \frac{N_R}{K} \times N_S$. Si tous les n-uplets de R peuvent être charger dans la mémoire le coût de réduit à $C = N_R + N_S$

Conclusion :
La Jointure par hachage est plus efficace que la jointure par boucles imbriquées dans le cas où les deux tables sont de grande taille.
L'inconvénient du hachage c'est qu'il n'est appliqué que pour la jointure avec égalité

V- L'optimisation : Les Vue Matérialisées

Les vues matérialisée (VM) permettent de créer des vues physiques d'une table ou d'une requête SQL sur un ensemble de table. La différence d'une vue standard c'est que les données sont dupliquées. On l'utilise pour, des fins d'optimisation de performance, lorsqu'une requête demande l'exécution d'un ensemble de sous-requêtes, ou pour faire des réplications de table.
La 'fraicheur' des données de la VM dépend des options choisies. Le décalage entre les données de la table maître et la VM peut être nul (rafraichissement synchrone) ou d'une durée planifiée : heure, jour,etc. Les vues matérialisées peuvent être utilisées directement par l'optimiseur, afin de modifier les chemins d'exécution des requêtes. Pour ce, il faut disposer du privilège QUERY REWRITE pour pouvoir réécrire la requête, et que QUERY_REWRITE_ENABLED soit TRUE
Syntaxe:
(ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE).

Les VMs restent, par conséquent, un moyen très simple pour répliquer et dupliquer les données d'une base de données dans d'autres base de données stockées dans des sites distants. Dans ce cas les mises à jour des opérations d'écriture seront effectuées en différé c-a-d les écritures seront validées dans la base de données où on a réalisé ces écritures avant d'être validées dans les sites distants.

V-1- Les types de vues matérialisées

Il existe deux types de VM :

- Les VMs sur clé primaire. C'est le type par défaut

CREATE MATERIALIZED VIEW NOM_VM
REFRESH with primary key
AS SELECT * FROM ….

- Les VMs sur rowid. Utile lorsque la vue ne contient pas de (ou pas toutes les colonnes de la) clé primaire

CREATE MATERIALIZED VIEW NOM_VM
REFRESH with rowid
AS SELECT * FROM ….

Contraintes (assez fortes !). Ce type de VM :

- Ne permet pas le fast refresh s'il n'a pas eu un complet avant.
- Est interdit si on a SELECT avec distinct, group by, connect by, fonctions d'aggregats, opérateurs ensemblistes, sous requêtes.

V-2- Les méthodes de rafraichissement de la VM

FAST : mise-à-jour partielle de la copie locale. C'est la méthode la plus efficace, elle utilise des journaux spécifiques traçant les modifications de la table maître. Lorsque des changements sont effectués sur les données des tables maîtres, Oracle stocke les lignes qui dérivent ces changements dans le journal de vue matérialisée (fichiers Log), et utilise ensuite ces changements dans le journal de vue matérialisé pour réactualiser les vues matérialisées par rapport à la table maître. Un journal de vues matérialisées est un objet de schéma qui enregistre les modifications apportées à une table de base afin qu'une vue matérialisée définie sur la table de base puisse être actualisée de manière incrémentielle. *Chaque journal de vues matérialisées est associé à une seule table de base*. Le journal des vues matérialisées réside dans la même base de données et le même schéma que sa table de base.

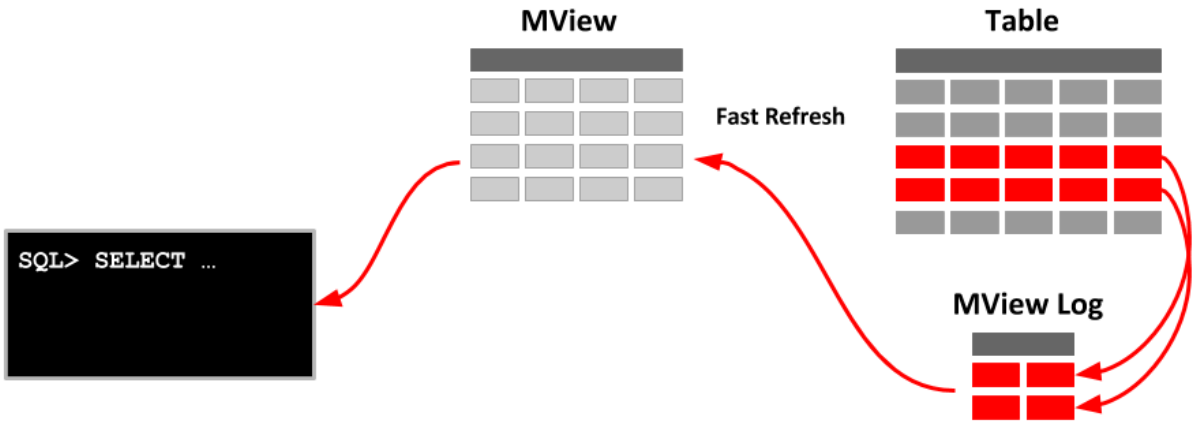


Figure-4 : Rafraichissement d'une VM en utilisant une journalisation stockée dans un fichier Log.

Syntaxe de création d'une journalisation sur une table:

```
create Materialized View Log on Table_name with Rowid, Primary
key (liste attributs) including new values;
```

with Rowid, Primary key: inclure la colonne de clé primaire et le rowids de toutes les lignes modifiées de la table de base dans un journal de vues matérialisées

Liste Attributs : la liste des attributs de la table qui sont utilisés par la requête de la VM

Including new values: permet à la BDD d'enregistrer, dans le journal des VMs, à la fois les anciennes et les nouvelles valeurs pour les opérations de mise à jour LMD. Concerne une table sur laquelle vous disposez d'une vue agrégée matérialisée à table unique

Conditions pour appliquer un rafraîchissement rapide sur sur VM

La requête de définition de la vue matérialisée est restreinte comme suit :

- La vue matérialisée ne doit pas contenir de références à des expressions non répétitives telles que `SYSDATE` et `ROWNUM`.
- La vue matérialisée ne doit pas contenir de références `RAW` ou `LONG RAW` de types de données.
- Il ne peut pas contenir de `SELECT` sous-requête de liste.
- Il ne peut pas contenir de fonctions analytiques (par exemple, `RANK`) dans la `SELECT` clause.
- Il ne peut pas contenir de `MODEL` clause.
- Il ne peut pas contenir de `HAVING` clause avec une sous-requête.
- Il ne peut pas contenir des requêtes imbriquées qui ont `ANY`, `ALL` ou `NOT EXISTS`.
- Il ne peut pas contenir de `[START WITH …] CONNECT BY` clause.
- Il ne peut pas contenir plusieurs tableaux détaillés sur différents sites.
- La vue matérialisée à la validation ne peut pas avoir de tables de détails distantes.
- Les vues matérialisées imbriquées doivent avoir une jointure ou un agrégat.

Conditions pour appliquer un rafraîchissement rapide sur les VMs avec jointures uniquement

La définition de requêtes pour des vues matérialisées avec des jointures uniquement et sans agrégats a les restrictions suivantes sur l'actualisation rapide :

- Toutes les restrictions générales sur le rafraîchissement rapide.
- Ils ne peuvent pas avoir de `GROUP BY` clauses ou d'agrégats.
- Les Rowids de toutes les tables de la `FROM` liste doivent apparaître dans la `SELECT` liste de la requête.
- Les journaux de vues matérialisées doivent exister avec des rowids pour toutes les tables de base de la `FROM` liste de la requête.

Conditions pour appliquer un rafraîchissement rapide sur les VMs avec agrégation

La définition de requêtes pour des vues matérialisées avec des agrégats ou des jointures a les restrictions suivantes sur l'actualisation rapide :

- Toutes les restrictions générales sur le rafraîchissement rapide.
- Toutes les tables de la vue matérialisée doivent avoir des journaux de vue matérialisée, et ces journaux doivent :
    - Contenir toutes les colonnes de la table référencée dans la vue matérialisée.
    - Spécifier avec `ROWID` et `INCLUDING NEW VALUES`.
    - Spécifier la `SEQUENCE` clause si la table doit contenir un mélange d'insertions/chargements directs, de suppressions et de mises à jour.
- Seuls `SUM`, `COUNT`, `AVG`, `STDDEV`, `VARIANCE`, `MIN` et `MAX` sont pris en charge pour une actualisation rapide.
- `COUNT(*)` doit être spécifié.
- Pour chaque agrégat tel que `AVG(expr)`, `SUM(expr)`, le correspondant `COUNT(expr)` doit être présent.
- Si `VARIANCE(expr)` ou `STDDEV(expr)` est spécifié `COUNT(expr)` et `SUM(expr)` doit être spécifié.
- La `SELECT` colonne de la requête de définition ne peut pas être une expression complexe avec des colonnes de plusieurs tables de base. Une solution de contournement possible consiste à utiliser une vue matérialisée imbriquée.
- La `SELECT` liste doit contenir toutes les `GROUP BY` colonnes.
- Si la vue matérialisée présente l'un des éléments suivants, l'actualisation rapide n'est prise en charge que sur les insertions.
    - Vues matérialisées avec `MIN` ou `MAX` agrégats
    - Vues matérialisées qui n'ont `SUM(expr)` pas `COUNT(expr)`
    - Vues matérialisées sans `COUNT(*)`

  Une telle vue matérialisée est appelée vue matérialisée à insertion uniquement.
- Une vue matérialisée avec `MAX` ou `MIN` est rapidement actualisable après suppression ou instructions DML mixtes si elle n'a pas de `WHERE` clause.
- S'il n'y a pas de jointures externes, vous pouvez avoir des sélections et des jointures arbitraires dans la `WHERE`clause.
- Les vues agrégées matérialisées avec jointures externes peuvent être actualisées rapidement après des charges DML conventionnelles et directes, à condition que seule la table externe ait été modifiée. En outre, des contraintes uniques doivent exister sur les colonnes de jointure de la table de jointure interne. S'il existe des jointures externes, toutes les jointures doivent être connectées par `AND` et doivent utiliser l'opérateur d' égalité (=).

- Pour les vues matérialisées avec `CUBE`, `ROLLUP`, groupes de regroupement ou concaténation de celles-ci, les restrictions suivantes s'appliquent :
    - La `SELECT` liste doit contenir un identificateur de regroupement qui peut être soit une `GROUPING_ID`fonction sur toutes les `GROUP BY`expressions, soit `GROUPING`une fonction pour chaque `GROUP BY`expression. Par exemple, si la `GROUP BY`clause de la vue matérialisée est " `GROUP BY CUBE(a, b)`", la `SELECT`liste doit contenir soit " `GROUPING_ID(a, b)`" soit " `GROUPING(a) AND GROUPING(b)`" pour que la vue matérialisée puisse être actualisée rapidement.
    - `GROUP BY` ne doit pas donner lieu à des regroupements en double. Par exemple, " `GROUP BY a, ROLLUP(a, b)`" n'est pas actualisable rapidement car il en résulte des regroupements en double " `(a), (a, b), AND (a)`"

Exemple :

```
create Materialized View Log on commandes with Rowid, Primary
key (idclient) including new values;


Create Materialized View V1
Refresh Fast on Commit as
select idclient, count(idcommande) as nb
from commandes
group by Idclient;
```

with Rowid, Primary key : inclure la colonne de clé primaire et le rowids de toutes les lignes modifiées de la table de base dans un journal de vues matérialisées

COMPLETE : effectue le refresh complet en exécutant le SELECT de définition de la MV. Un rafraîchissement complet peut être demandé à tout moment au cours de la vie de toute vue matérialisée. L'actualisation implique la relecture des tableaux détaillés pour recalculer les résultats de la vue matérialisée. Cela peut être un processus très long, surtout s'il y a d'énormes quantités de données à lire et à traiter. Par conséquent, on doit toujours tenir compte du temps nécessaire pour traiter une actualisation complète avant de la demander.

Il existe cependant des cas où la seule méthode d'actualisation disponible pour une vue matérialisée déjà créée est l'actualisation complète car la vue matérialisée ne satisfait pas aux conditions spécifiées ci-dessous pour une actualisation rapide (consulter https://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm)

Quelques restrictions générales sur le rafraichissement Rapide

- La vue matérialisée ne doit pas contenir de références à des expressions non répétitives telles que SYSDATEet ROWNUM.
- Ne peut pas contenir de HAVING clause avec une sous-requête.

- Ne peut pas contenir de [START WITH …] CONNECT BYclause.
- Ne peut pas contenir plusieurs tableaux détaillés sur différents sites.
- La vue matérialisée lors de la validation ne peut pas avoir de tables de détails distantes.

Restrictions sur l'actualisation rapide sur les vues matérialisées avec jointures uniquement et sans agrégats

- Toutes les restrictions générales mentionnées ci-dessus.
- Ils ne peuvent pas avoir de GROUP BYclauses ou d'agrégats.
- Les Rowids de toutes les tables de la FROM liste doivent apparaître dans la SELECT liste de la requête.
- Les journaux de vues matérialisées doivent exister avec des rowids pour toutes les tables de base de la FROM liste de la requête.

Quelques Restrictions sur l'actualisation rapide des vues matérialisées avec des agrégats

- Toutes les restrictions générales mentionnées ci-dessus.
- Seulement SUM, COUNT, AVG, STDDEV, VARIANCE, MINet MAXsont pris en charge pour le rafraîchissement rapide.
- COUNT(*) doit être spécifié.
- Pour chaque agrégat tel que AVG(expr), le correspondant COUNT(expr)doit être présent. Oracle recommande que SUM(expr)soit spécifié.

FORCE : **Cette méthode** effectue dans un premier temps une actualisation rapide (FAST). Si cette dernière échoue, une actualisation complète se produira.

Vérification de la méthode de rafraîchissement sur une VM :
Oracle dispose de la procédure dbms_Mview.explain_Mview('NomVM') qui permet de vérifier la méthode d'actualisation qui peut être appliquée sur une VM déjà créé. Cette procédure donne même les causes du non fonctionnement d'une méthode d'actualisation (Fast ou Complet).
Pour utiliser cette procédure il faut tout d'abord créer la table MV_CAPABILITIES

```
create table MV_CAPABILITIES_TABLE
(
  statement_id      varchar(30) ,
  mvowner           varchar(30) ,
  mvname            varchar(30) ,
  capability_name   varchar(30) ,
  possible          character(1) ,
  related_text      varchar(2000) ,
  related_num       number ,
  msgno             integer ,
  msgtxt            varchar(2000) ,
```

```
  seq                 number
) ;
```

On exécute ensuite la procédure

```
begin
dbms_Mview.explain_Mview('NomVM');
end;
```

On peut afficher les données de la table MV_Capabilities_table :

```
SELECT capability_name, possible,
FROM mv_capabilities_table
WHERE capability_name like '%FAST%';
```

CAPABILITY_NAME	POSSIBLE
REFRESH FAST	N
REFRESH FAST AFTER INSERT	N
REFRESH FAST AFTER ONETAB DML	N
REFRESH FAST AFTER ANY DML	N
REFRESH FAST PCT	N

V-3- Les modes de rafraichissement de la VM

ON COMMIT: synchrone, rafraîchissement lorsqu'une transaction modifiant les tables maîtresses fait un commit. Ce mode de rafraichissement est utilisé que si la table maître est dans la même base de données où on a créé la vue matérialisée. Par conséquent le mode « on commit » n'est pas pris en charge dans les bases de données distantes.

ON DEMAND *asynchrone (par défaut):* C'est un rafraîchissement sur demande de l'usager ou à un instant qui peut être spécifié avec les clauses START et NEXT. Ce mode est utilisé dans le cas des VM qui utilisent des tables maitresses distantes. L'actualisation peut être effectuée avec les méthodes d'actualisation fournies dans le `DBMS_SYNC_REFRESH` ou les `DBMS_MVIEW` packages :

- `DBMS_MVIEW.REFRESH`

  Actualisez une ou plusieurs vues matérialisées.

- `DBMS_MVIEW.REFRESH_ALL_MVIEWS`

  Actualisez toutes les vues matérialisées.

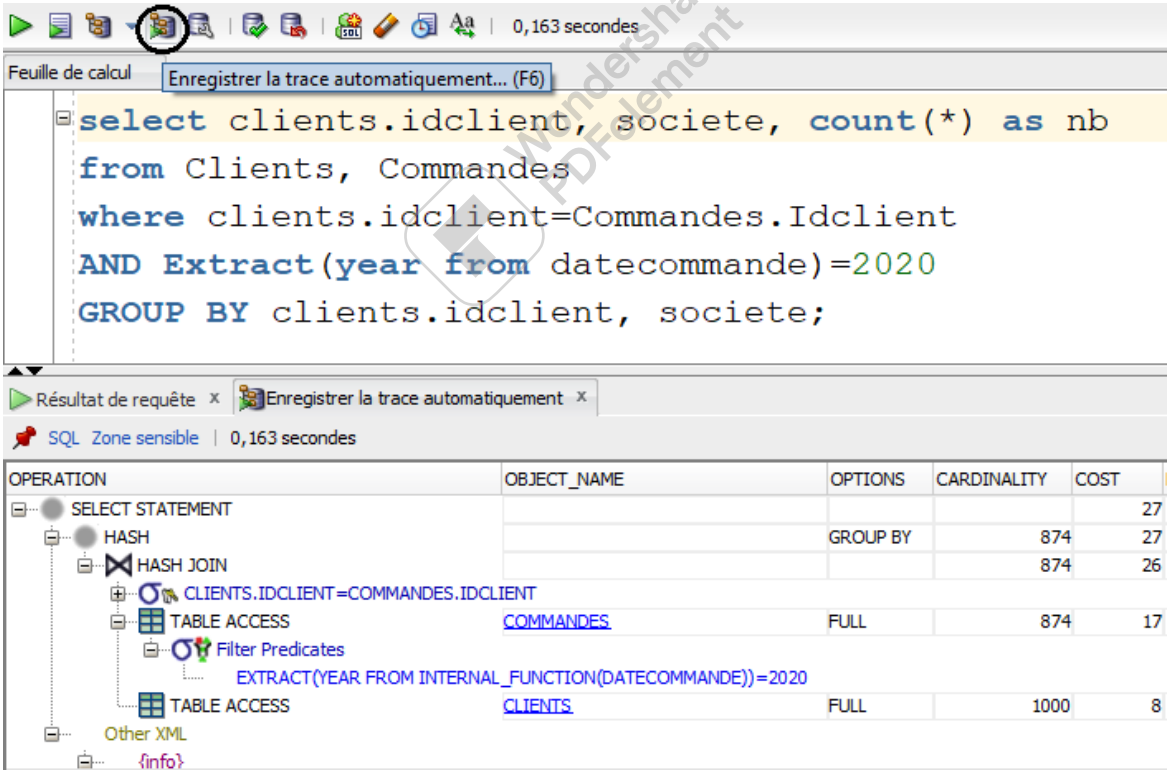
- `DBMS_MVIEW.REFRESH_DEPENDENT`

Actualisez toutes les vues matérialisées qui dépendent d'une table principale ou d'une vue matérialisée ou d'une liste de tables principales ou de vues matérialisées spécifiées.

FOR UPDATE: permet de mettre à jour la copie locale (les changements sont propagés aux tables maîtresses).

VI- Plan d'exécution d'Oracle

Pour chaque requête SQL, Oracle permet d'afficher le plan d'exécution choisi par l'optimiseur. On observe ainsi toutes les opérations faites par le moteur de requête. « SQL-developper » nous donne la possibilité d'afficher ce plan d'une manière simple en cliquant sur « enregistrer la trace automatique » ou en tapant F6. On notera que la lecture se fait du bas vers le haut : Ainsi dans l'exemple ci-dessous le moteur commence par effectuer la restriction sur la table commande en gardant que les tuples où l'attribut « datecommande » correspond à 2020 puis joindre ces enregistrements avec la table client et enfin regrouper par « idclient » et « societe ».



On notera les indicateurs suivants pour chaque ligne du plan :

- **Cost** est un indicateur de coût de calcul de la requête à ce stade (exemple : le coût de l'opération de balayage de la table clients est de 8). Cet indicateur est une synthèse estimée des différents coûts (accès disque, CPU, etc.). Cet indicateur est sans unité puisqu'il ne sert qu'à comparer différents plans entre eux.

- **Cardinality** est une estimation a priori du nombre de lignes remontées par la requête à ce stade (exemple : le nombre de lignes remontées par la jointure est de 874).
- **Operation** : Le nom de l'opération que peut effectuer l'optimiseur pour joindre les tables (ex : 'NESTED LOOPS', 'HASH JOIN' ou MERGE JOIN).
- **Option :** représente le mode pour accéder aux objets consultés. On note les modes

    - **Table Access FULL :** Parcours séquentiel (balayage complet)
    - **Table Access by Rowid :** Accès direct par adresse
    - **Index (Inique|Range|…) Scan** : Accès par index
    - **Table Access Hash :** Accès par hachage
    - **Table Access Cluster :** Accès par cluster